How to Quantize Neural Networks with TensorFlow

When modern neural networks were being developed, the biggest challenge was getting them to work at all! That meant that accuracy and speed during training were the top priorities. Using floating point arithmetic was the easiest way to preserve accuracy, and GPUs were well-equipped to accelerate those calculations, so it's natural that not much attention was paid to other numerical formats.

These days, we actually have a lot of models being deployed in commercial applications. The computation demands of training grow with the number of researchers, but the cycles needed for inference expand in proportion to users. That means pure inference efficiency has become a burning issue for a lot of teams.

That is where quantization comes in. It's an umbrella term that covers a lot of different techniques to store numbers and perform calculations on them in more compact formats than 32-bit floating point. I am going to focus on eight-bit fixed point, for reasons I'll go into more detail on later.

Why does Quantization Work?

Training neural networks is done by applying many tiny nudges to the weights, and these small increments typically need floating point precision to work (though there are research efforts to use quantized representations here too).

Taking a pre-trained model and running inference is very different. One of the magical qualities of deep networks is that they tend to cope very well with high levels of noise in their inputs. If you think about recognizing an object in a photo you've just taken, the network has to ignore all the CCD noise, lighting changes, and other non-essential differences between it and the training examples it's seen before, and focus on the important similarities instead. This ability means that they seem to treat low-precision calculations as just another source of noise, and still produce accurate results even with numerical formats that hold less information.

Why Quantize?

Neural network models can take up a lot of space on disk, with the original AlexNet being over 200 MB in float format for example. Almost all of that size is taken up with the weights for the neural connections, since there are often many millions of these in a single model. Because they're all slightly different floating point numbers, simple compression formats like zip don't compress them well. They are arranged in large layers though, and within each layer the weights tend to be normally distributed within a certain range, for example -3.0 to 6.0.

The simplest motivation for quantization is to shrink file sizes by storing the min and max for each layer, and then compressing each float value to an eight-bit integer representing the closest real number in a linear set of 256 within the range. For example with the -3.0 to 6.0 range, a 0 byte would represent -3.0, a 255 would stand for 6.0, and 128 would represent about 1.5. I'll go into the exact calculations later, since there's some subtleties, but this means you can get the benefit of a file on disk that's shrunk by 75%, and then convert back to float after loading so that your existing floating-point code can work without any changes.

Another reason to quantize is to reduce the computational resources you need to do the inference calculations, by running them entirely with eight-bit inputs and outputs. This is a lot more difficult since it requires changes everywhere you do calculations, but offers a lot of potential rewards. Fetching eight-bit values only requires 25% of the memory bandwidth of floats, so you'll make much better use of caches and avoid bottlenecking on RAM access. You can also typically use SIMD operations that do many more operations per clock cycle. In some case you'll have a DSP chip available that can accelerate eight-bit calculations too, which can offer a lot of advantages.

Moving calculations over to eight bit will help you run your models faster, and use less power (which is especially important on mobile devices). It also opens the door to a lot of embedded systems that can't run floating point code efficiently, so it can enable a lot of applications in the IoT world.

Why Not Train in Lower Precision Directly?

There have been some experiments training at lower bit depths, but the results seem to indicate that you need higher than eight bit to handle the back propagation and gradients. That makes implementing the training more complicated, and so starting with inference made sense. We also already have a lot of float models already that we use and know well, so being able to convert them directly is very convenient.

How Can You Quantize Your Models?

TensorFlow has production-grade support for eight-bit calculations built it. It also has a process for converting many models trained in floating-point over to equivalent graphs using quantized calculations for inference. For example, here's how you can translate the latest GoogLeNet model into a version that uses eight-bit computations:

```
curl http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tg
tar xzf /tmp/inceptionv3.tgz -C /tmp/
bazel build tensorflow/tools/quantization/tools:quantize_graph
bazel-bin/tensorflow/tools/quantization/tools/quantize_graph \
  --input=/tmp/classify_image_graph_def.pb \
  --output_node_names="softmax" --output=/tmp/quantized_graph.pb \
  --mode=eightbit
```

This will produce a new model that runs the same operations as the original, but with eight bit calculations internally, and all weights quantized as well. If you look at the file size, you'll see it's about a quarter of the original (23MB versus 91MB). You can still run this model using exactly the same inputs and outputs though, and you should get equivalent results. Here's an example:

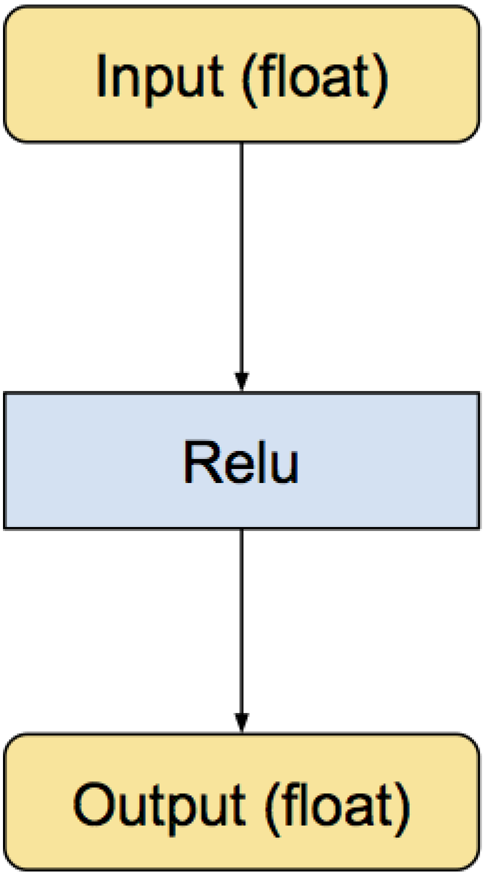
```
# Note: You need to add the dependencies of the quantization operation to the
#       cc_binary in the BUILD file of the label_image program:
#
#     //tensorflow/contrib/quantization:cc_ops
#     //tensorflow/contrib/quantization/kernels:quantized_ops

bazel build tensorflow/examples/label_image:label_image
bazel-bin/tensorflow/examples/label_image/label_image \
--image=<input-image> \
--graph=/tmp/quantized_graph.pb \
--labels=/tmp/imagenet_synset_to_human_label_map.txt \
--input_width=299 \
--input_height=299 \
--input_mean=128 \
--input_std=128 \
--input_layer="Mul:0" \
--output_layer="softmax:0"
```

You'll see that this runs the newly-quantized graph, and outputs a very similar answer to the original.

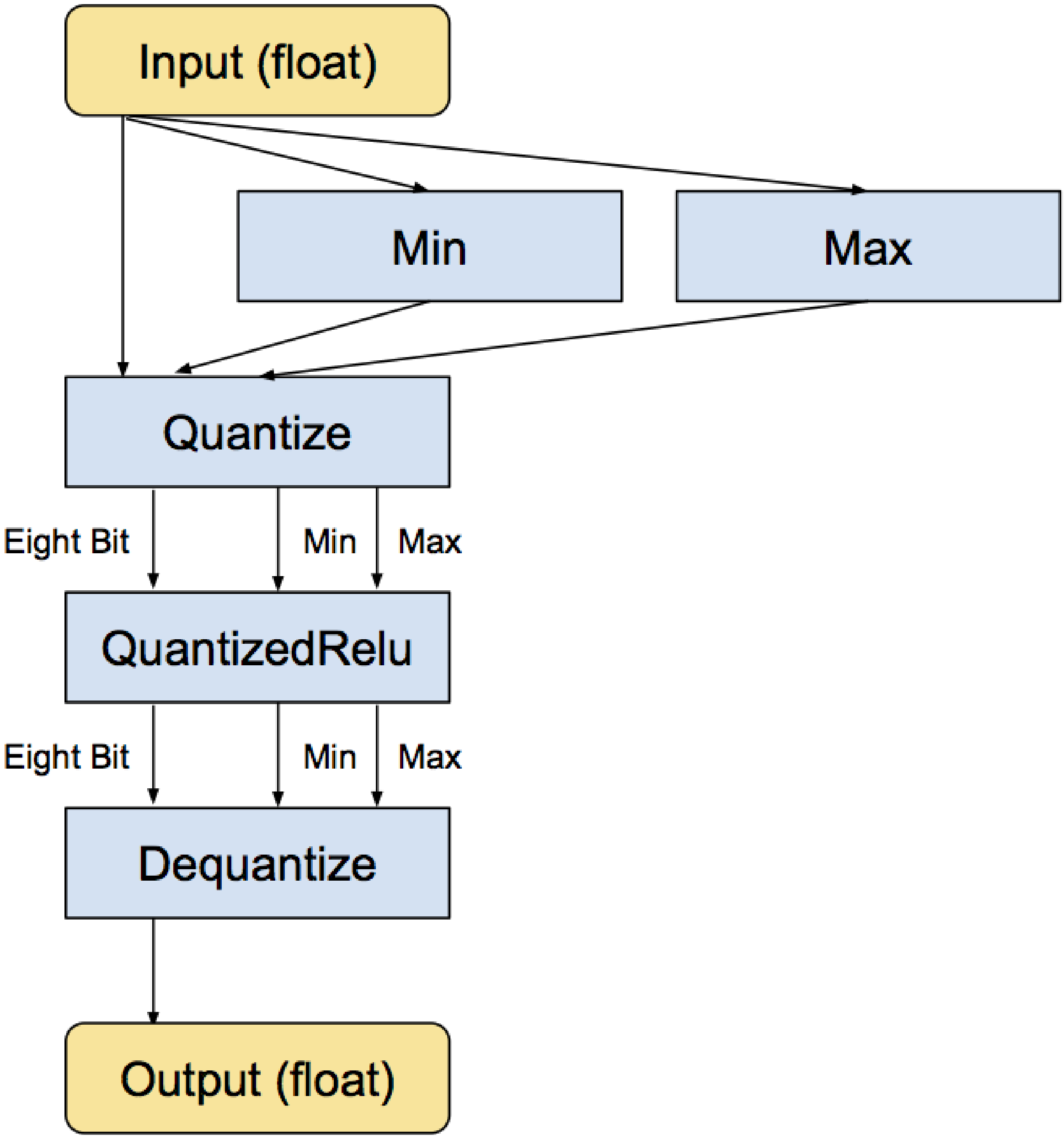
You can run the same process on your own models saved out as GraphDefs, with the input and output names adapted to those your network requires. I recommend that you run them through the freeze_graph script first, to convert checkpoints into constants stored in the file.

How Does the Quantization Process Work?

We've implemented quantization by writing equivalent eight-bit versions of operations that are commonly used during inference. These include convolution, matrix multiplication, activation functions, pooling operations and concatenation. The conversion script first replaces all the individual ops it knows about with quantized equivalents. These are small sub-graphs that have conversion functions before and after to move the data between float and eight-bit. Below is an example of what they look like. First here's the original Relu operation, with float inputs and outputs:

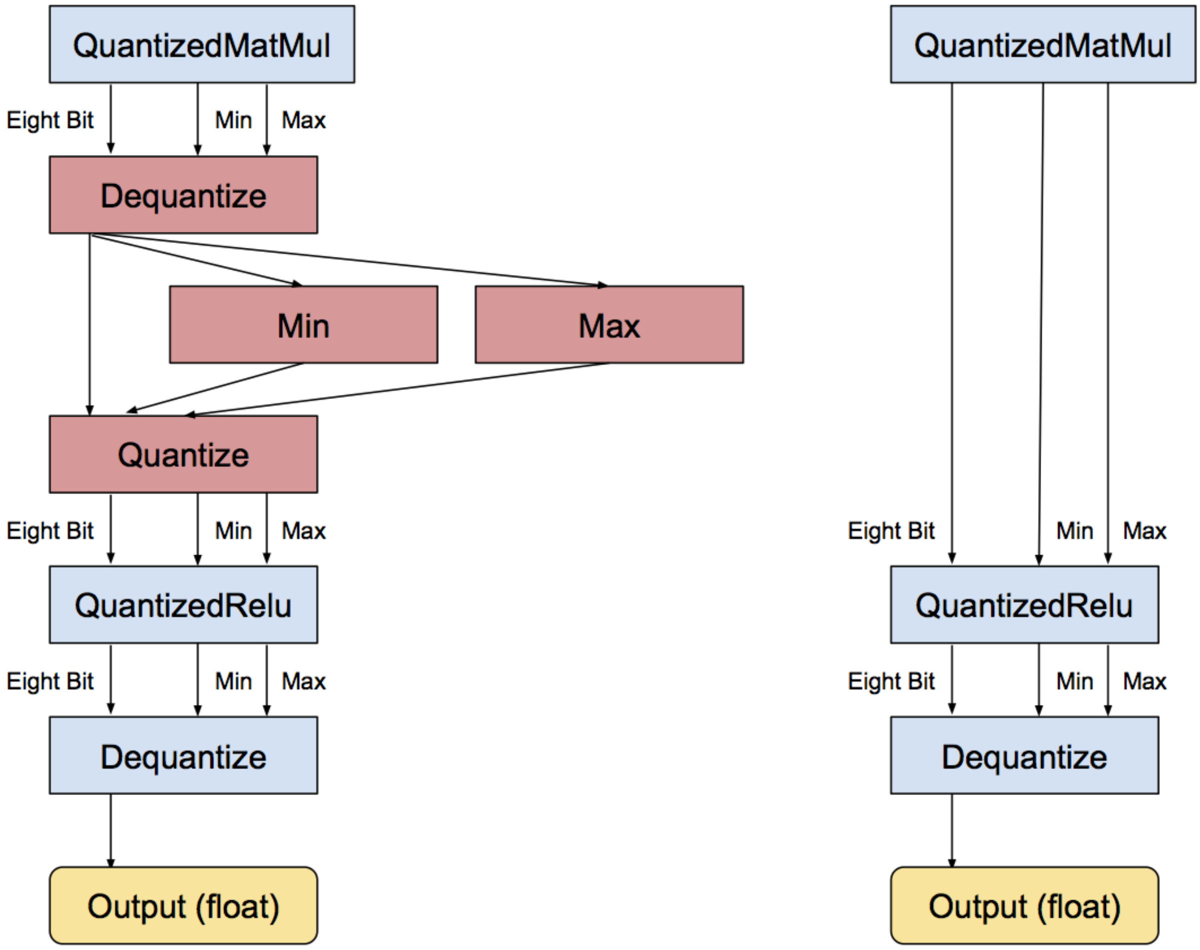


Then, this is the equivalent converted subgraph, still with float inputs and outputs, but with internal conversions so the calculations are done in eight bit.



The min and max operations actually look at the values in the input float tensor, and then feeds them into the Dequantize operation that converts the tensor into eight-bits. There's more details on how the quantized representation works later on.

Once the individual operations have been converted, the next stage is to remove unnecessary conversions to and from float. If there are consecutive sequences of operations that all have float equivalents, then there will be a lot of adjacent Dequantize/Quantize ops. This stage spots that pattern, recognizes that they cancel each other out, and removes them, like this:



Applied on a large scale to models where all of the operations have quantized equivalents, this gives a graph where all of the tensor calculations are done in eight bit, without having to convert to float.

What Representation is Used for Quantized Tensors?

We approach converting floating-point arrays of numbers into eight-bit representations as a compression problem. We know that the weights and activation tensors in trained neural network models tend to have values that are distributed across comparatively small ranges (for example you might have -15 to +15 for weights, -500 to 1000 for activations on an image model, though the exact numbers will vary). We also know from experiment that neural nets tend to be very robust in the face of noise, and so the noise-like error produced by quantizing down to a small set of values will not hurt the precision of the overall results very much. We also want to pick a representation that's easy to perform calculations on, especially the large matrix multiplications that form the bulk of the work that's needed to run a model.

These led us to pick a representation that has two floats to store the overall minimum and maximum values that are represented by the lowest and highest quantized value. Each entry in the quantized array represents a float value in that range, distributed linearly between the minimum and maximum. For example, if we have minimum = -10.0, and maximum = 30.0f, and an eight-bit array, here's what the quantized values represent:

```
Quantized | Float
--------- | -----
0         | -10.0
255       | 30.0
128       | 10.0
```

The advantages of this format are that it can represent arbitrary magnitudes of ranges, they don't have to be symmetrical, it can represent signed and unsigned values, and the linear spread makes doing multiplications straightforward. There are alternatives like Song Han's code books (http://arxiv.org/pdf/1510.00149.pdf) that can use lower bit depths by non-linearly distributing the float values across the representation, but these tend to be more expensive to calculate on.

The advantage of having a strong and clear definition of the quantized format is that it's always possible to convert back and forth from float for operations that aren't quantization-ready, or to inspect the tensors for debugging purposes. One implementation detail in TensorFlow that we're hoping to improve in the future is that the minimum and maximum float values need to be passed as separate tensors to the one holding the quantized values, so graphs can get a bit dense!

The nice thing about the minimum and maximum ranges is that they can often be pre-calculated. Weight parameters are constants known at load time, so their ranges can also be stored as constants. We often know the ranges for inputs (for examples images are usually RGB values in the range 0.0 to 255.0), and many activation functions have known ranges too. This can avoid having to analyze the outputs of an operation to determine the range, which we need to do for math ops like convolution or matrix multiplication which produce 32-bit accumulated results from 8-bit inputs.

What's Next?

We've found that we can get extremely good performance on mobile and embedded devices by using eight-bit arithmetic rather than floating-point. You can see the framework we use to optimize matrix multiplications at gemmlowp (https://github.com/google/gemmlowp). We still need to apply all the lessons we've learned to the TensorFlow ops to get maximum performance on mobile, but we're actively working on that. Right now, this quantized implementation is a reasonably fast and accurate reference implementation that we're hoping will enable wider support for our eight-bit models on a wider variety of devices. We also hope that this demonstration will encourage the community to explore what's possible with low-precision neural networks.

上次更新日期：三月 8, 2017